

A1 Assignment – ESE516-SPRING 2019

DUE DATE: MONDAY FEBRUARY 4TH 2019 before 11:59pm EST (By almost midnight). To be submitted on Canvas

Remember: Please submit your Atmel Studio 7 project in a zip file to canvas. Also, please upload a *.docx with answers to the concept questions.

A1 Assignment Description

1.) Understanding the Starter Code – answering questions [50 points]

Please download the starter code available at Google Drive:

https://drive.google.com/open?id=1irDo_EkpbkH52q3HgkWzI8XPCYMFfmcU

The starter code is an Atmel Studio 7 project for your SAMW25 that does the following:

- Initialize the UART at 115200 8N1
- Reads characters using interrupts from the Uart and add it to a ringbuffer
 - You can read a character with the function “int SerialConsoleReadCharacter(uint8_t *rxChar)”
 - The system echoes the characters the user writes, so they can read what they are typing!
- Writes characters from a ringbuffer
 - You can send characters by adding them to the buffer using the function “void SerialConsoleWriteString(char * string)”

Once you downloaded the code, please go through the code. In particular, focus on how the device is reading characters and writing characters. It is very similar to what we did in class!

Please answer the following questions and SUBMIT IN A WORD DOCUMENT WITH YOUR NAME:

- 1.) What does "InitializeSerialConsole()" do? In said function, what is "cbufRx" and "cbufTx"?**
- 2.) How are "cbufRx" and "cbufTx" initialized? Where is the library that defines them (please tell the *C file they come from).**
- 3.) Where are the character arrays where the RX and TX characters are being stored on at the end?**
- 4.) Please draw a diagram that explain the program flow for UART reception – starting at the user typing a character and ending on how that characters ends up in the circular buffer "cbufRx". Please make reference to specific functions in the starter code.**
- 5.) Please draw a diagram that explain the program flow for the UART transmission – Starting from a string added by the program to the circular buffer "cbufTx" and ending on characters being shown on the screen of a PC (On Teraterm, for example). Please make reference to specific functions in the starter code.**

2.) Programming Warm-Up: Debug Logger Module [50 points]

What is a Debug Logger Module:

The Debug Logger Module will be C calls that we can later use in the development of our IoT products to help us debug as well as indicate any system information. They are useful for the programmer to write human-readable strings to a terminal to determine the state of a device. It is the same idea as when we use “print” statements on code running on a PC to print statements about the program.

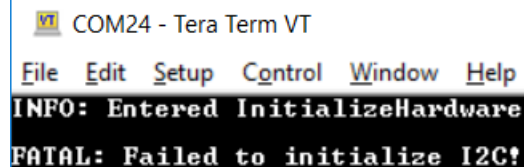
For example, imagine you are implementing a very complicated state machine along with some hardware initialization. With the use of the Logger Module, you could make calls that print to a terminal the current state of the program. Here is an example of how that can be useful:

```
//Attempts to initialize I2C Hardware
void InitializeHardware(void)
{
    int error = ERROR_NONE;

    LogMessage(LOG_INFO_LVL, "Entered InitializeHardware\r\n"); //<Logs message to Terminal

    error = InitI2c(void); //Initializes I2C devices
    if(error != ERROR_NONE)
    {
        LogMessage(LOG_FATAL_LVL, "Failed to initialize I2C!\r\n"); //<Logs message to Terminal
    }else
    {
        LogMessage(LOG_INFO_LVL, "Initialized I2C!\r\n"); //<Logs message to Terminal
    }
}
```

An example output if the previous code failed would be the following:



The screenshot shows a terminal window titled "COM24 - Tera Term VT". The menu bar includes "File", "Edit", "Setup", "Control", "Window", and "Help". The output text is as follows:

```
INFO: Entered InitializeHardware
FATAL: Failed to initialize I2C!
```

Your Task: Implement the Debug Logger Module, that when called, prints the message to the terminal.
The DebugLogger has these different levels of messages:

```
enum eDebugLogLevels {
    LOG_INFO_LVL = 0,    //Logs an INFO + message
    LOG_DEBUG_LVL = 1,   //Logs a DEBUG + message
    LOG_WARNING_LVL = 2, //Logs a WARNING + message
    LOG_ERROR_LVL = 3,   //Logs an ERROR + message
    LOG_FATAL_LVL = 4,   //Logs a FATAL + message (a non-recoverable error)
    LOG_OFF_LVL = 5,     //Enum to indicate levels are off
    N_DEBUG_LEVELS = 6   //Max number of log levels
};
```

Our Debug Logger should be able to just print messages that are above or below a set level given (and set) by the following functions:

```
eDebugLogLevels getLogLevel(void); //Returns the current logging level

void setLogLevel(eDebugLogLevels debugLevel) //Sets the Debug Logger level to the given level
```

Implement the following function, which has the following inputs:

```
void LogMessage(enum eDebugLogLevels level, const char *format, ...)
```

*enum eDebugLevels: Determines the log levels of the message to output. If the level is smaller than the current “logLevel” it is not printed.

* `const char *format`: Pointer to a array of characters to be printed.

* Variable List: the “...” in C denotes a variable list. Please refer to <https://www.cprogramming.com/tutorial/c/lesson17.html> for more information. In this argument, we expect the variables that you would normally use in a vsprintf (please see example on https://www.tutorialspoint.com/c_standard_library/c_function_vsprintf.htm).

Usage example:

```
setLogLevel(LOG_ERROR_LVL); //Sets the Debug Logger to only allow messages with LOG_ERROR_LVL or higher to be printed

LogMessage(LOG_INFO_LVL, "Performing Temperature Test...\r\n", //This should NOT print

LogMessage(LOG_FATAL_LVL, "Error! Temperature over %d Degrees!\r\n", sensorTemperature); //This should print
```

TIPS

***Check the function “SerialConsoleWriteString”. You can use that to print to the TX buffer**

*** As said before, check https://www.tutorialspoint.com/c_standard_library/c_function_vsprintf.htm to understand how to use vsprintf for the variable argument list.**

3.) Command Line Interface [100 points]

Implement a Command Line Interface. The interface must:

- Receive characters from a serial terminal program via UART
- Execute a command when the “enter” key is pressed (Tip: What is the ENTER key’s value in ASCII?)
- When “help\n” is received from the terminal, it replies with the list of available functions that the user can type
- When the user types one of these functions, the MCU executes the function and returns the appropriate data
- If the user types the wrong command, it returns “ERROR\n”
- If the user pushes the back-key, it erases a character.
 - If we send the character ASCII for backspace through the terminal, it will erase a character on the Terminal’s side, so the erase can be visualized.

The following page mentions the commands to implement. For any variables, such as Firmware version numbers, use dummy data.

Set of Commands to Implement

1. **help**

- a. Description: Prints all the available commands and a short synopsis
- b. Inputs: None
- c. Output: List of all available commands and their descriptions

2. **ver_bl**

- . Description: Prints the bootloader firmware version
- a. Inputs: None
- b. Outputs: String in the format MAJOR.MINOR.PATCH
- c. Notes:
 - i. Use MAJOR.MINOR.PATCH versioning, per <https://semver.org/>
 - ii. Store each number as an unsigned 8-bit integer
 - iii. Example output: 1.4.88, where MAJOR = 1, MINOR = 4, PATCH = 88

3. **ver_app**

- . Description: Prints the application code firmware version
- a. Inputs: None
- b. Outputs: String in the format MAJOR.MINOR.PATCH
- c. Notes:
 - . Use MAJOR.MINOR.PATCH versioning, per <https://semver.org/>
 - i. Store each number as an unsigned 8-bit integer
 - ii. Example output: 1.4.88, where MAJOR = 1, MINOR = 4, PATCH = 88

4. **mac**

- a. Description: returns the mac address of the device
- b. Inputs: None
- c. Outputs: mac address in standard form ff.ff.ff.ff.ff, where each octet is in hex.
- d. Notes: The mac address resides with the ATWINC chip so you will need to query it.
- e. Example:
- f. **This function only needs to return dummy info for now.**

5. **ip**

- a. Description: returns the ip address of the device in the standard format: ex. 255.255.255.255
- b. Inputs: None
- c. Outputs: current IP assigned to the device.
- d. Notes: The IP address resides with the ATWINC chip so you will need to query it for the address.
- e. Example:
- f. **This function only needs to return dummy info for now. You can return 255.255.255.255 always**

6. **devName**

- g. Description: Returns the name of the developer (your name)
- h. Inputs: None
- i. Outputs: Your name, in ASCII, or nickname
- j. Notes:
- k. **This function only needs to return dummy info for now. If you don't want to return your name, put your**

pennKey

7. **setDeviceName <string name>**

- a. Description: Sets the name of the device to the given string
- b. Inputs: A string (separater from "setDeviceName" by a space
- c. Output: "Device name set to <string name>"
 - 1. Example:

```
setDeviceName Dave
Device name set to Dave
```

3. **getDeviceName**

- a. Description: Gets the name of the device to the given string
- b. Inputs: None
- c. Output: "Device name is '<Device's name>'"
 - 1. Example:

```
getDeviceName
device name is ESE516
```